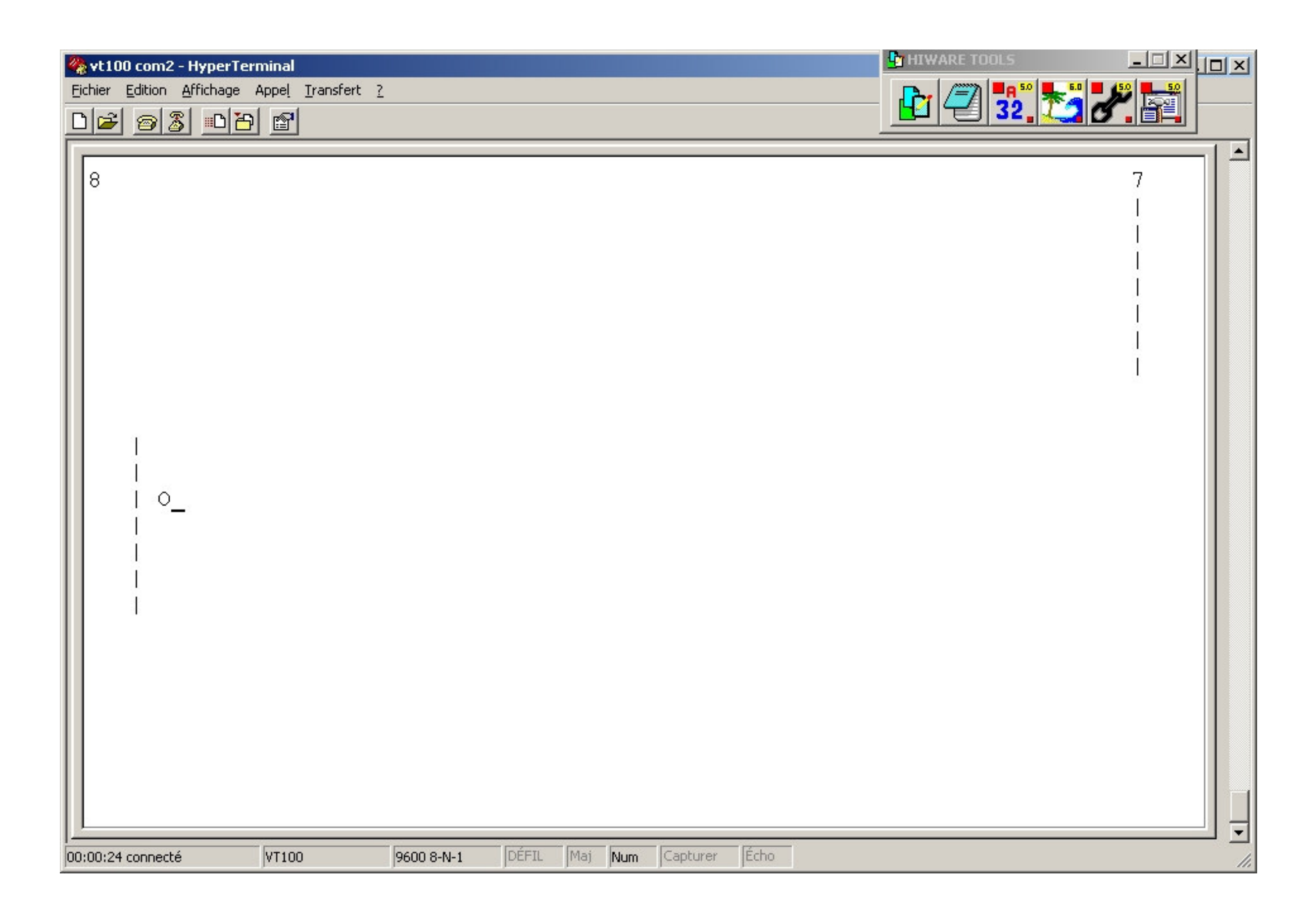


Mini-projet assembleur

Pong 68k



Professeur responsable :

SAVATON Guillaume

Etudiants I1 :

FRANÇOIS Sébastien
BARON Julien

Sommaire

1) Introduction

1.1) But du mini-projet

Découvrir la programmation en assembleur d'un microprocesseur 68000 par le développement de librairies, mises par la suite en œuvre dans un projet.

La première librairie contient la gestion de chaînes de caractères, celles-ci étant représentées en mémoire par une suite d'octets (code ASCII) terminée par un caractère nul. Nous avions besoin, pour pouvoir gérer du texte dans le projet, de mettre au point différentes fonctions nous permettant de créer et de modifier une chaîne de caractères.

La deuxième librairie gère quant à elle le périphérique de communication série intégré à la cible. Elle le configure et gère l'envoi et la réception d'octets échangés avec l'ordinateur au moyen d'hyper terminal. Elle utilise un mécanisme d'interruption pour la réception d'un caractère. Nous aurons aussi besoin d'une fonction permettant d'envoyer directement une chaîne de caractères.

2) Conception des librairies

2.1) str.asm

Cette librairie fournit l'ensemble des fonctions de gestion de chaînes d'octets.

strln	Calcule la longueur d'une chaîne
strcpy	Copie une chaîne
strcat	Concatène deux chaînes
strcmp	Compare deux chaînes
itoa	Convertit un entier en sa représentation ASCII
sprintf	Ecrit dans une chaîne selon un format donné

2.2) sci.asm

Cette librairie fournit l'ensemble des fonctions nécessaires à l'utilisation du port série de la cible 68000.

sci_init	Configure le port (vitesse, vecteur d'interruption pour la réception d'un octet)
sci_send_byte	Envoie un octet
sci_has_byte	Test si un octet est en attente dans le tampon d'entrée
sci_receive_byte	Lit un octet reçu
sci_send_ascii_string	Transmet une chaîne terminée par nul
sci_receive_ascii_string	Reçoit une chaîne
sci_enable_rx_irq	Active l'interruption de réception d'octet
sci_disable_rx_irq	Désactive " "

3) Spécifications du projet

3.1) Description

Nous avons choisi de réaliser un jeu inspiré du célèbre jeu Pong, parmi les premières réalisations ludiques dans l'histoire de l'informatique.



Ping-Pong sur Odyssey de Magnavox (1972), précurseur du Pong d'Atari

3.2) Démarche

Nous choisi de réutiliser dans un premier temps une partie du code de l'étape 3, dans laquelle trois balles parcouraient le terminal en rebondissant sur les bords.

Nous avons décidé de nous appuyer sur cette gestion du rebond d'une balle pour la réutiliser dans notre Pong, en ajoutant le rebond sur les raquettes et en gardant la possibilité d'augmenter aisément le nombre de raquettes et de balles.

Nous avons alors commencé à développer le code de gestion d'une raquette, puis des collisions avec celle-ci. Durant ce développement, nous avions prévu la possibilité d'intégrer facilement une seconde raquette en utilisant systématiquement un adressage indirect indexé sur l'indice du joueur.

Enfin, nous avons mis en place la gestion d'un compteur, incrémenté lorsque la balle rebondit sur le mur opposé.

3.3) Mode d'emploi du jeu

Le jeu se joue à deux, les touches sont les suivantes :

	Joueur 1	Joueur 2
Monter la barre	A	P
Descendre la barre	Q	M

La balle ne sort jamais, il n'y a pas de remise en jeu. Elle rebondit tout simplement sur les murs verticaux en incrémentant le compteur adverse.
Il faut essayer de renvoyer la balle de l'autre côté, et surtout éviter que la balle ne passe entre le mur et la barre, car elle rebondirait plusieurs fois entre la barre et le mur et à chaque rebond le compteur sera incrémenté.

4) Outils

4.1) L'éditeur de code

Nous avons très vite voulu chercher une alternative à l'édition des fichiers dans Notepad. Après avoir utilisé GVim, nous avons installé Scite, un éditeur libre supportant aussi la coloration syntaxique de l'assembleur du 68000.

http://scintilla.sourceforge.net/SciTEDownload.html

4.2) Le Makefile

Nous avons choisi de rédiger un makefile pour le projet, de manière à ne pas avoir à assembler les fichiers manuellement à chaque modification.

Il se présente sous la forme suivante :

```
JEU_SRCS = jeu/jeu.asm jeu/barre.asm jeu/balle.asm jeu/collision.asm
jeu/curseur.asm jeu/compteur.asm

JEU_PRM  = jeu/jeu.prm

SCI_SRCS = sci/sci.asm

STR_SRCS = str/str.asm

STARTUP_SRCS = startup/startup.asm

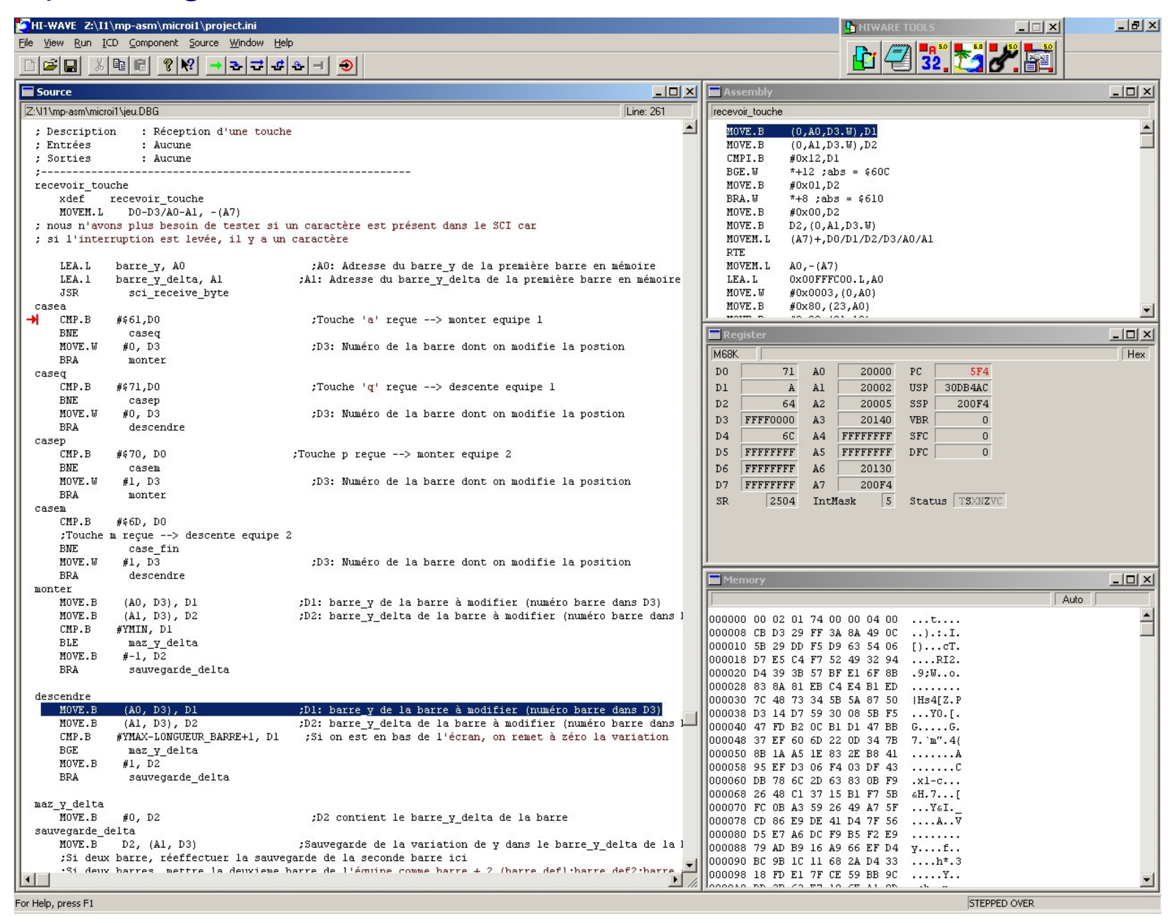
makeall: makesci makestr makestartup
    $(ASM) $(JEU_SRCS)
    $(LINK) $(JEU_PRM)
makesci:
        $(ASM) $(SCI_SRCS)
makestr:
        $(ASM) $(STR_SRCS)
makestartup:
        $(ASM) $(STARTUP_SRCS)
```

Ce makefile appelé sans paramètre assemble à nouveau chaque fichier et effectue la liaison. Nous avons essayé d'écrire un makefile avec une règle implicite pour indiquer comment transformer un fichier .asm en .o, sans succès :

```
Top: Z:\I1\mp-asm\microi1\jeu.mak

>> in "Z:\I1\mp-asm\microi1\jeu.mak", line 17, col 0, pos 325
%.o: %.asm
^
ERROR M5010: Illegal line
*** 1 error(s), 0 warning(s), 0 information message(s) ***
*** command line: 'jeu.mak ' ***
*** Error occured while processing! ***
```

4.3) Le débogueur



Exemple d'utilisation du débogueur :

Un point d'arrêt a été placé dans la fonction d'interruption de réception de caractère. Une touche a donc été frappée sur le terminal, son code est en D0 (0x71 soit 'q'), la barre du joueur 1 doit donc descendre, le microprocesseur va changer la valeur de barre_y_delta du joueur 1 puis quitter la fonction d'interruption. La boucle principale se chargera de redessiner la barre par la suite.

5) Conception : liste et description des fichiers

5.1) cste.inc

Ce fichier d'entête contient l'ensemble des constantes définissant la taille des barres, leurs coordonnées initiales, l'espace de jeu.
Les autres fichiers asm sont systématiquement accompagnés d'un .inc associé contenant les XREF.

5.2) balle.asm

Ce fichier contient l'ensemble du code gérant le déplacement des balles ainsi que leurs rebonds.

afficher_balle	Dessine la balle passée en paramètre
effacer_balle	Efface la balle passée en paramètre
maj_balle	Calcule les nouvelles coordonnées de la balle et tiens compte des rebonds sur les murs

5.3) barre.asm

Ce fichier contient l'ensemble du code gérant le déplacement des barres ainsi que leurs rebonds.

afficher_barre	Dessine la barre passée en paramètre
effacer_barre	Efface la barre passée en paramètre

5.4) collision.asm

Ce fichier contient l'ensemble du code gérant les rebonds sur les barres et les murs verticaux

calcul_position_futur	Teste si la balle va taper sur une barre ou un mur. Si oui, met à jour les vecteurs vitesse pour la faire rebondir

5.5) compteur.asm

Ce fichier contient l'ensemble du code gérant les compteurs.

afficher_cpt	Dessine le compteur passé en paramètre

5.6) curseur.asm

Ce fichier contient l'ensemble du code gérant le déplacement du curseur du terminal VT100.

placer_curseur	Place le curseur aux coordonnées passées en paramètre

5.7) jeu.asm

Ce fichier contient le code principal de l'application ainsi que la définition des variables globales.

main	Fonction principale, initialisation suivie de la boucle principale infinie
recevoir_touche	Fonction d'interruption gérant l'appui sur des touches

6) Algorithmes employés

6.1) Le SCI et le terminal VT100

Cette partie explique comment nous utilisons notre périphérique de communication RS232 (SCI) pour communiquer avec un terminal de type VT100.

Nous pouvons envoyer au terminal des caractères pour lui indiquer qu'il doit se préparer à recevoir une commande (séquence d'échappement), suivi de la commande à proprement parler.

Nous n'utilisons que deux des multiples commandes d'un terminal VT100.

La première nous est fort utile tout au long du programme. Elle consiste à placer le curseur à l'endroit que l'on souhaite sur l'écran du terminal. Pour cela, nous utilisons la séquence **[P;PH**
Remplacer les P par respectivement la position en abscisse et la position en ordonnée du curseur.

La seconde consiste à effacer entièrement l'écran avec la séquence : **[PJ**
Voici les différentes possibilités d'utilisation de cette commande : (remplacer la lettre P par le numéro correspondant à l'une d'entre elles) :

0 Effacement à partir du curseur, jusqu'à la fin de l'écran
1 Effacement du début de l'écran jusqu'au curseur
2 Effacement de tout l'écran (pas de déplacement du curseur).

Lors de cet effacement, on souhaite remettre le curseur à sa position initiale : (0 ; 0).
Pour cela, nous utilisons la séquence **[0;0H** qui nous remet à la position d'origine.

6.2) Les barres

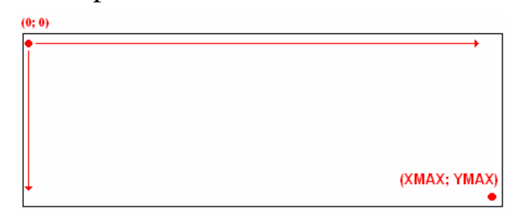
6.2.1) Structure

Chaque barre (ou raquette) est représentée par sa position dans la fenêtre (abscisse et ordonnée) et un entier indiquant la variation d'ordonnée que subit la barre.

Dans jeu.asm :

```
typdef struct s_barre
{
 int8 x, y ;
 int8 y_delta ; //>0 == barre descend ; <0 == barre monte
}t_barre ;
```

Le choix de prendre un y_delta > 0 pour une barre qui descend vient de la configuration de l'écran. Pour trouver la nouvelle position de la barre, il suffira donc d'additionner l'ancienne position avec les valeurs de son delta.



Origines des coordonnées dans le terminal

6.2.2) Affichage

L'affichage ne peut se faire par rafraîchissement complet et systématique de l'écran, car à 9600bauds, cela serait bien trop lent. Nous optimiserons donc celui-ci en ne transmettant que les modifications à apporter à l'affichage et ne dessinons les barres que lors de leur déplacement. Toujours pour optimiser, nous ne réaffichons pas toute la barre, mais seulement la partie qui a changé.

Dans jeu.asm :

```
void maj_affichage_barre(t_barre *barres)
{
    int8 i ;
    for (i = 0 ; i < NB_BARRES ; i++)
    {
        if (barre[i]->y_delta != 0)
        {
            if (barre[i]->y + barre[i]->y_delta < YMIN ||
                    barre[i]->y + barre[i]->y_delta >= YMAX)
            {
                  effacer_barre(barre[i]);
                  afficher_barre(barre[i]);
                  if (barre[i]->y_delta != 0)
                      barre[i]->y += (barre[i]->y_delta > 0 ? 1 : -1);
            }
            barre[i]->y_delta = 0;
        }
    }
}
```

Dans barre.asm :

```
void effacer_barre(t_barre barre)
{
    if (barre.y_delta == 0) //Barre ne bouge pas
        return;
    else if (barre.y_delta < 0) //Barre monte->Effacement du caractère du bas
        placerCurseur(barre.x, barre.y + TAILLE_BARRE - 1);
    else //Barre descend->Effacement du caractère du haut
        placerCurseur(barre.x, barre.y);
    afficherRS232(' ') ;  //Effacement par affichage d'un espace
}

void afficher_barre(t_barre barre)
{
    if (barre.y_delta == 0)
        return;
    else if (barre.y_delta < 0) //Barre monte->Affichage d'un caractère au dessus de la
barre
        placerCurseur(barre.x, barre.y - 1);
    else //Barre descend->Affichage d'un caractère au dessous de la barre
        placerCurseur(barre.x, barre.y + TAILLE_BARRE);
    afficherRS232(CARACTERE_BARRE);
}
```

Dans le programme assembleur, le label maj_affichage_barre n'est pas une fonction, elle fait partie intégrante de la boucle principale du programme.
Les tests ont été réalisés avec et sans l'optimisation d'affichage qui consiste à ne réafficher que ce qui a changé (haut et bas de la barre). La différence est flagrante. Si nous devions tout réafficher à chaque fois, nous verrions à l'écran que la balle serait considérablement ralentie, ce qui montre bien que l'envoi d'un caractère prend du temps (bridé dans un premier temps par la liaison 9600 bauds et ensuite seulement par la vitesse du microprocesseur).

6.3) La balle

Cette partie est un peu plus complexe puisqu'il faut gérer les rebonds de la balle sur les bords de l'écran et sur les raquettes. Il s'agit donc de mettre en place un contrôle de collision.

6.3.1) Structure

Chaque balle (parce qu'il est aisé d'en mettre plusieurs pour compliquer le jeu) est représentée par 4 arguments : ses coordonnées (abscisse et ordonnée) et ses vecteurs vitesses (composante en x et composante en y).

Dans jeu.asm :

```
typedef struct s_balle
{
 int8 x, y ;            // Compris entre XMIN et XMAX, YMIN et XMAX
 int8 vx, vy ;          // Compris entre -1 et 1
}
```

6.3.2) Gestion des collisions

Le problème est simple, dès que la barre rencontre un mur vertical ou une raquette, il faut changer la direction de la composante en x de son vecteur vitesse. Dès qu'elle rencontre un mur horizontal, il faut inverser la composante y de son vecteur vitesse.

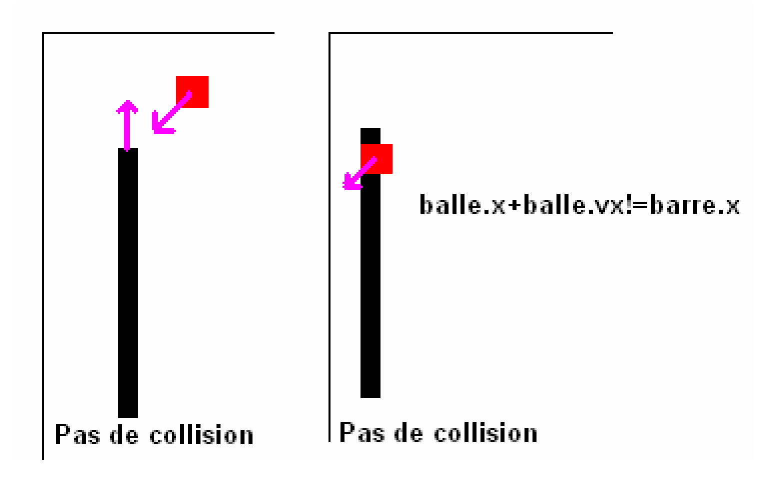
Dans balle.asm :

```
void maj_balle(t_balle *balle, t_barre *barre)
{
 int8 i, j ;
 boolean MAJ ;
 for (i = 0 ; i < NB_BALLES ; i++)
 {
        MAJ = 1 ;
        //Collision avec un mur
        if (balle[i]->x + balle[i]->vx < XMIN)
        {
               balle[i]->vx = -balle[i]->vx ;
               MAJ = 0 ;
        }
        else if (balle[i]->x + balle[i]->vx >= XMAX)
        {
               balle[i]->vx = -balle[i]->vx ;
               MAJ = 0 ;
        }
        if (balle[i]->y + balle[i]->vy < YMIN)
        {
               balle[i]->vy = -balle[i]->vy ;
               MAJ = 0 ;
        }
        else if (balle[i]->y + balle[i]->vy >= YMAX)
        {
               balle[i]->vy = -balle[i]->vy ;
               MAJ = 0 ;
        }
        if (MAJ)
        {
               balle[i]->x += balle[i]->vx ;
               balle[i]->y += balle[i]->vy ;
        }
        //Collision avec une raquette
        calcul_position_futur(&balle[i], barre);
 }
}
```

Dans collision.asm :

```
void calcul_position_futur(t_balle *balle, t_barre *barre)
{
 int8 i ;
 for (i = 0 ; i < NB_BARRES ; i++)
 {
 //Si au tour suivant balle et raquette ont même abscisse
 //C'est que l'on est sur les abscisses précédant la raquette
        if (balle->x + balle->vx == barre[i].x)
        {
                //Balle devant la raquette->On la renvoie
                if (balle->y >= barre[i].y && balle->y < barre[i].y + TAILLE_BARRE)
                        balle->vx = -balle->vx ;
        }
 }
}
```

Dans la majorité des cas, notre code fonctionne correctement, mais un bug d'affichage peut tout de même arriver. Celui-ci étant difficile à expliquer par écrit, je vais tenter l'explication par un schéma.



On voit sur ce schéma que lorsque la balle arrive par exemple au-dessus de la barre, il n'y a pas collision. Mais si au tour suivant, la barre monte, la balle va s'afficher par-dessus la barre, puis continuer à avancer puisque aucune collision n'aura été détectée et le programme effacera donc le caractère de la barre.

6.4) Déroulement du jeu

Tout le jeu se déroule dans une seule et même fonction qui tourne en boucle indéfiniment. Elle remet à jour la position de la balle et celle des barres.
Le y_delta des barres est mis à jour sur une interruption du SCI, donc on ne réaffiche les barres que de temps en temps, puisque l'on teste dans maj_affichage_barre si y_delta est différent de 0.

```
void main(void)
{
 t_balle *balles ;
 t_barre *barres ;
 init_reg() ; // Initialisation du périphérique SCI (registre, niveau d'interruption…)
 init_balle(balles) ;
 init_barre(barres) ;
 while (1)     // main_boucle
 {
        effacerBalle(balles) ;
        maj_balle(balles, barres) ;
        afficher_balle(balles) ;
        maj_affichage_barre(barres) ;
 }
}
```

7) Etat d'avancement

Les objectifs des étapes 0 à 3 ont été atteints, et nous avons un jeu simple qui fonctionne de manière satisfaisante.

Etape	Avancement
Etape 0 Prise en main	**COMPLÈTE**
Etape 1 Bibliothèque STR	**COMPLÈTE**
Etape 2 Bibliothèque SCI	**COMPLÈTE**
Etape 3 Gestion d'interruption dans SCI	**COMPLÈTE**
Etape 4 Incrustation d'horloge	**NÉANT**
Etape 5 Développement de Pong 68k	**COMPLET**

8) Conclusion

Nous avons trouvé que le mini-projet était un complément essentiel au cours de microprocesseur proposé en I1. Il pourrait même être intéressant de le commencer plus tôt de manière à pouvoir bénéficier du cours et dans la même semaine de l'application concrète d'écriture et déboguage de code.

Nous avons apprécié de nous investir dans un projet que nous avions choisi. Nous sommes rapidement parvenus à obtenir un programme ludique ce qui était motivant !

Même si le développement sur microprocesseur 68000 n'est plus vraiment d'actualité, ce projet nous a permis de voir bien en détails le fonctionnement de deux points techniques intéressants : la mise en œuvre de l'instruction LINK et le passage d'un nombre d'arguments variable (équivalent du va_list en C).

Nous suggérons de remettre en état les cibles cassées pour l'année prochaine, et d'y fixer des câbles, afin que ces détails matériels n'entravent pas le déroulement du mini-projet.